FUNCTIONAL PEARL
Weaving a Web

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

JOHAN JEURING
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* johanj@cs.uu.nl)

Just a little bit of it can bring you up and down.
— Genesis, *it*

1 Introduction

Suppose, you want to implement a structured editor for some term type, so that the user can navigate through a given term and perform edit actions on subterms. In this case you are immediately faced with the problem of how to keep track of the cursor movements and the user's edits in a reasonably efficient manner. In a previous pearl G. Huet (1997) introduced a simple data structure, the *Zipper*, that addresses this problem—we will explain the Zipper briefly in Sec. 2. A drawback of the Zipper is that the type of cursor locations depends on the structure of the term type, that is, each term type gives rise to a different type of locations (unless you are working in an untyped environment). In this pearl we present an alternative data structure, the *Web*, that serves the same purpose but that is parametric in the underlying term type. Sec. 3 – 6 are devoted to the new data structure. Before we unravel the Zipper and explore the Web, let us first give a taste of their use.

The following (excerpt of a) term type for representing programs in some functional language serves as a running example.[1]

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Term} & = & \mathit{Var}\ \mathit{String} \\ & | & \mathit{Abs}\ \mathit{String}\ \mathit{Term} \\ & | & \mathit{App}\ \mathit{Term}\ \mathit{Term} \\ & | & \mathit{If}\ \mathit{Term}\ \mathit{Term}\ \mathit{Term} \end{array}$$

In fact, the term type has been chosen so that we have constructors with no, one, two

[1] The programs are given in the functional programming language Haskell 98 (Peyton Jones & Hughes, 1999).

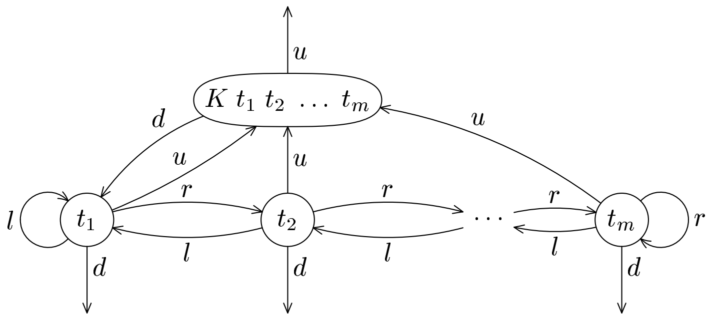


Fig. 1. Navigating through the term $K\ t_1\ t_2\ \ldots\ t_m$.

and three recursive components. Here is an example element of *Term*, presumably the right-hand side of the definition of the factorial function:

```
rhs = Abs "n" (If (App (App (Var "=") (Var "n")) (Var "0"))
                  (Var "1")
                  (App (App (Var "+") (Var "n"))
                       (App (Var "fac") (App (Var "pred") (Var "n"))))).
```

But ouch, the program contains a typo: in the else branch the numbers are added rather than multiplied. To correct the program let us use the Zipper library. It supplies a type of locations, four navigation primitives, a function that starts the navigation taking a term into a location and a function that extracts the subterm at the current location:

```
Loc                     :: *
top                     :: Term → Loc
down, up, left, right   :: Loc → Loc
it                      :: Loc → Term.      -- record label
```

Note that *it* is a record label so that we can use Haskell's record syntax to change a subterm: $l\{it = t\}$ replaces the subterm at location l by t. The navigation primitives have the following meaning: *down* goes to the leftmost child (or rather, the leftmost recursive component) of the current node, *up* goes to the parent, *left* goes to the left sibling and *right* goes to the right sibling. Fig. 1 illustrates the navigation primitives.

The following session with the Haskell interpreter Hugs (Jones & Peterson, 1999) shows how to correct the definition of the factorial function (a location is displayed by showing the associated subterm; $$ always refers to the previous value).

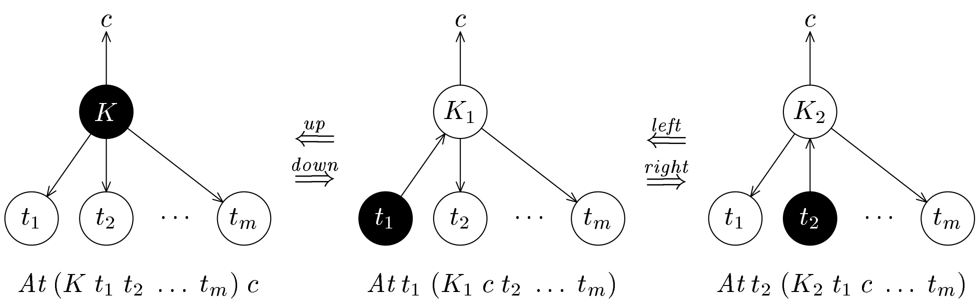
```
> top rhs
Abs "n" (If (App (App (Var "=") (Var "n")) (Var "0")) (...))
> down $$
If (App (App (Var "=") (Var "n")) (Var "0")) (Var "1") (...)
> down $$
App (App (Var "=") (Var "n")) (Var "0")
```

```
>    right $$
Var "1"
>    right $$
App (App (Var "+") (Var "n")) (App (Var "fac") (App (Var "pred") (Var "n")))
>    down $$
App (Var "+") (Var "n")
>    down $$
Var "+"
>    $$ {it = Var "*"}
Var "*"
>    up $$
App (Var "*") (Var "n")
```

We go down twice to the first argument of *If*, then move two times to the right into the else branch, where we again go down twice. As to be expected the local change is remembered when we go up. In a real editor the edit actions are most likely more advanced, but such advanced edit actions usually consist of combinations of primitive actions like the ones used in the session above.

2 The Zipper

The Zipper is based on pointer reversal. If we follow a pointer to a subterm, the pointer is reversed to point from the subterm to its parent so that we can go up again later. A location is simply a pair $At\ t\ c$ consisting of the current subterm t and a pointer c to its parent. The upward pointer corresponds to the *context* of the subterm. It can be represented as follows. For each constructor K that has m recursive components we introduce m context constructors $K_1, \ldots, K_m$. Now, consider the location $At\ (K\ t_1\ t_2\ \ldots\ t_m)\ c$. If we go down to t_1, we are left with the context $K \bullet t_2\ \ldots\ t_m$ and the old context c. To represent the combined context, we simply plug c into the hole to obtain $K_1\ c\ t_2\ \ldots\ t_m$. Thus, the new location is $At\ t_1\ (K_1\ c\ t_2\ \ldots\ t_m)$. The following picture illustrates the idea (the filled circle marks the current cursor position).



The implementation of the Zipper for the datatype *Term* is displayed in Fig. 2. Clearly, the larger the term type the larger the context type and the larger the implementation effort for the navigation primitives.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Loc} & = & \mathit{At}\{\mathit{it} :: \mathit{Term}, \mathit{ctx} :: \mathit{Ctx}\} \\
\textbf{data}\ \mathit{Ctx} & = & \mathit{Top} \\
 & \mid & \mathit{Abs}_1\ \mathit{String}\ \mathit{Ctx} \\
 & \mid & \mathit{App}_1\ \mathit{Ctx}\ \mathit{Term} \\
 & \mid & \mathit{App}_2\ \mathit{Term}\ \mathit{Ctx} \\
 & \mid & \mathit{If}_1\ \mathit{Ctx}\ \mathit{Term}\ \mathit{Term} \\
 & \mid & \mathit{If}_2\ \mathit{Term}\ \mathit{Ctx}\ \mathit{Term} \\
 & \mid & \mathit{If}_3\ \mathit{Term}\ \mathit{Term}\ \mathit{Ctx} \\
\mathit{down}, \mathit{up}, \mathit{left}, \mathit{right} & :: & \mathit{Loc} \to \mathit{Loc} \\
\mathit{down}\ (\mathit{At}\ (\mathit{Var}\ s)\ c) & = & \mathit{At}\ (\mathit{Var}\ s)\ c \\
\mathit{down}\ (\mathit{At}\ (\mathit{Abs}\ s\ t_1)\ c) & = & \mathit{At}\ t_1\ (\mathit{Abs}_1\ s\ c) \\
\mathit{down}\ (\mathit{At}\ (\mathit{App}\ t_1\ t_2)\ c) & = & \mathit{At}\ t_1\ (\mathit{App}_1\ c\ t_2) \\
\mathit{down}\ (\mathit{At}\ (\mathit{If}\ t_1\ t_2\ t_3)\ c) & = & \mathit{At}\ t_1\ (\mathit{If}_1\ c\ t_2\ t_3) \\
\mathit{up}\ (\mathit{At}\ t\ \mathit{Top}) & = & \mathit{At}\ t\ \mathit{Top} \\
\mathit{up}\ (\mathit{At}\ t_1\ (\mathit{Abs}_1\ s\ c)) & = & \mathit{At}\ (\mathit{Abs}\ s\ t_1)\ c \\
\mathit{up}\ (\mathit{At}\ t_1\ (\mathit{App}_1\ c\ t_2)) & = & \mathit{At}\ (\mathit{App}\ t_1\ t_2)\ c \\
\mathit{up}\ (\mathit{At}\ t_2\ (\mathit{App}_2\ t_1\ c)) & = & \mathit{At}\ (\mathit{App}\ t_1\ t_2)\ c \\
\mathit{up}\ (\mathit{At}\ t_1\ (\mathit{If}_1\ c\ t_2\ t_3)) & = & \mathit{At}\ (\mathit{If}\ t_1\ t_2\ t_3)\ c \\
\mathit{up}\ (\mathit{At}\ t_2\ (\mathit{If}_2\ t_1\ c\ t_3)) & = & \mathit{At}\ (\mathit{If}\ t_1\ t_2\ t_3)\ c \\
\mathit{up}\ (\mathit{At}\ t_3\ (\mathit{If}_3\ t_1\ t_2\ c)) & = & \mathit{At}\ (\mathit{If}\ t_1\ t_2\ t_3)\ c \\
\mathit{left}\ (\mathit{At}\ t\ \mathit{Top}) & = & \mathit{At}\ t\ \mathit{Top} \\
\mathit{left}\ (\mathit{At}\ t_1\ (\mathit{Abs}_1\ s\ c)) & = & \mathit{At}\ t_1\ (\mathit{Abs}_1\ s\ c) \\
\mathit{left}\ (\mathit{At}\ t_1\ (\mathit{App}_1\ c\ t_2)) & = & \mathit{At}\ t_1\ (\mathit{App}_1\ c\ t_2) \\
\mathit{left}\ (\mathit{At}\ t_2\ (\mathit{App}_2\ t_1\ c)) & = & \mathit{At}\ t_1\ (\mathit{App}_1\ c\ t_2) \\
\mathit{left}\ (\mathit{At}\ t_1\ (\mathit{If}_1\ c\ t_2\ t_3)) & = & \mathit{At}\ t_1\ (\mathit{If}_1\ c\ t_2\ t_3) \\
\mathit{left}\ (\mathit{At}\ t_2\ (\mathit{If}_2\ t_1\ c\ t_3)) & = & \mathit{At}\ t_1\ (\mathit{If}_1\ c\ t_2\ t_3) \\
\mathit{left}\ (\mathit{At}\ t_3\ (\mathit{If}_3\ t_1\ t_2\ c)) & = & \mathit{At}\ t_2\ (\mathit{If}_2\ t_1\ c\ t_3) \\
\mathit{right}\ (\mathit{At}\ t\ \mathit{Top}) & = & \mathit{At}\ t\ \mathit{Top} \\
\mathit{right}\ (\mathit{At}\ t_1\ (\mathit{Abs}_1\ s\ c)) & = & \mathit{At}\ t_1\ (\mathit{Abs}_1\ s\ c) \\
\mathit{right}\ (\mathit{At}\ t_1\ (\mathit{App}_1\ c\ t_2)) & = & \mathit{At}\ t_2\ (\mathit{App}_2\ t_1\ c) \\
\mathit{right}\ (\mathit{At}\ t_2\ (\mathit{App}_2\ t_1\ c)) & = & \mathit{At}\ t_2\ (\mathit{App}_2\ t_1\ c) \\
\mathit{right}\ (\mathit{At}\ t_1\ (\mathit{If}_1\ c\ t_2\ t_3)) & = & \mathit{At}\ t_2\ (\mathit{If}_2\ t_1\ c\ t_3) \\
\mathit{right}\ (\mathit{At}\ t_2\ (\mathit{If}_2\ t_1\ c\ t_3)) & = & \mathit{At}\ t_3\ (\mathit{If}_3\ t_1\ t_2\ c) \\
\mathit{right}\ (\mathit{At}\ t_3\ (\mathit{If}_3\ t_1\ t_2\ c)) & = & \mathit{At}\ t_3\ (\mathit{If}_3\ t_1\ t_2\ c) \\
\mathit{top} & :: & \mathit{Term} \to \mathit{Loc} \\
\mathit{top}\ t & = & \mathit{At}\ t\ \mathit{Top}
\end{array}
$$

Fig. 2. The zipper data structure for *Term*.

3 The Web

If you use the Web, the implementation effort is considerably smaller. All you have to do is to define a function that weaves a web. For the *Term* datatype it reads:

$$
\begin{array}{lcl}
\mathit{weave} & :: & \mathit{Term} \to \mathit{Weaver}\ \mathit{Term} \\
\mathit{weave}\ (\mathit{Var}\ s) & = & \mathit{con}_0\ \mathit{weave}\ (\mathit{Var}\ s) \\
\mathit{weave}\ (\mathit{Abs}\ s\ t_1) & = & \mathit{con}_1\ \mathit{weave}\ (\mathit{Abs}\ s)\ t_1 \\
\mathit{weave}\ (\mathit{App}\ t_1\ t_2) & = & \mathit{con}_2\ \mathit{weave}\ \mathit{App}\ t_1\ t_2 \\
\mathit{weave}\ (\mathit{If}\ t_1\ t_2\ t_3) & = & \mathit{con}_3\ \mathit{weave}\ \mathit{If}\ t_1\ t_2\ t_3.
\end{array}
$$

For each constructor K that has m recursive components we call the combinator con_m supplied by the Web library[2]. It takes $m+2$ arguments: the weaving function itself, a so-called *constructor function* and the m recursive components of K. Given m recursive components the constructor function builds a term that has K as the top-level constructor. So, if K only has recursive components (like *App* and *If*), then the constructor function is simply K. Otherwise, it additionally incorporates the non-recursive components of K.

The weaving function can be mechanically generated from a given datatype definition—so that you can use the Web even if you don't read the following sections. The equation for a constructor K takes the following general form

$$weave\ (K\ a_1\ \ldots\ a_n) \ = \ con_m\ weave\ (\lambda t_1\ \ldots\ t_m \to K\ a_1\ \ldots\ a_n)\ t_1\ \ldots\ t_m,$$

where the variables $\{t_1, \ldots, t_m\} \subseteq \{a_1, \ldots, a_n\}$ mark the recursive components of the constructor K.

The navigation primitives are the same as before except that the type of locations is now parametric in the underlying term type.

$$\begin{array}{lll} Loc & :: & \star \to \star \\ down, up, left, right & :: & Loc\ a \to Loc\ a \\ it & :: & Loc\ a \to a \qquad \text{-- record label} \end{array}$$

The weaving primitives are

$$\begin{array}{lll} Weaver & :: & \star \to \star \\ con_0 & :: & (a \to Weaver\ a) \to (a) \to Weaver\ a \\ con_1 & :: & (a \to Weaver\ a) \to (a \to a) \to a \to Weaver\ a \\ con_2 & :: & (a \to Weaver\ a) \to (a \to a \to a) \to a \to a \to Weaver\ a \\ con_3 & :: & (a \to Weaver\ a) \to (a \to a \to a \to a) \to a \to a \to a \to Weaver\ a \\ explore & :: & (a \to Weaver\ a) \to a \to Loc\ a. \end{array}$$

To turn a term t into a location one calls *explore weave* t—this is the only difference to the Zipper where we used *top* t.

The implementation is presented in three steps. Sec. 4 shows how to implement a web that allows you to navigate through a term without being able to change it. Sec. 5 describes the amendments necessary to support editing. Finally, Sec. 6 shows how to implement the interface above.

4 A Read-only Web

The idea underlying the Web is quite simple: given a term t we generate a graph whose nodes are labelled with subterms of t. There is a directed edge between two nodes t_i and t_j if one can move from t_i to t_j using one of the navigation primitives. The local structure of the graph is displayed in Fig. 1. A location is now a node

[2] Since Haskell currently has no support for defining variadic functions, the Web library only supplies con_0, ..., con_{max} where *max* is some fixed upper bound. This is not a limitation, however, since one can use as a last resort a function that operates on lists, see Exercise 1.

together with its outgoing edges; it is represented by the following datatype.

$$\begin{aligned}\textbf{data}\ \mathit{Loc}\ a = \mathit{At}\{ & \mathit{it} :: a,\\ & \mathit{down} :: \mathit{Loc}\ a,\\ & \mathit{up} :: \mathit{Loc}\ a,\\ & \mathit{left} :: \mathit{Loc}\ a,\\ & \mathit{right} :: \mathit{Loc}\ a\}\end{aligned}$$

The function *top* turns a term into a location.

$$\begin{aligned}&\mathit{top} && :: \mathit{Term} \to \mathit{Loc}\ \mathit{Term}\\ &\mathit{top}\ t && = r\ \textbf{where}\ r = \mathit{At}\ t\ (\mathit{weave}\ r\ t)\ r\ r\ r\end{aligned}$$

If the user goes down, the function *weave* is invoked, which lazily constructs the nodes of the web (in fact, this version of the web relies on lazy evaluation). It takes two arguments, a location and the label of the location, and yields the location of the first recursive component. If there is none, it simply returns the original location. Note that since we are working towards a solution, this version of *weave* does not yet have the type given in the previous section.

$$\begin{aligned}&\mathit{weave} && :: \mathit{Loc}\ \mathit{Term} \to \mathit{Term} \to \mathit{Loc}\ \mathit{Term}\\ &\mathit{weave}\ l_0\ (\mathit{Var}\ s) && = l_0\\ &\mathit{weave}\ l_0\ (\mathit{Abs}\ s\ t_1) && = l_1\\ &\quad \textbf{where}\ l_1 && = \mathit{At}\ t_1\ (\mathit{weave}\ l_1\ t_1)\ l_0\ l_1\ l_1\\ &\mathit{weave}\ l_0\ (\mathit{App}\ t_1\ t_2) && = l_1\\ &\quad \textbf{where}\ l_1 && = \mathit{At}\ t_1\ (\mathit{weave}\ l_1\ t_1)\ l_0\ l_1\ l_2\\ &\qquad\qquad l_2 && = \mathit{At}\ t_2\ (\mathit{weave}\ l_2\ t_2)\ l_0\ l_1\ l_2\\ &\mathit{weave}\ l_0\ (\mathit{If}\ t_1\ t_2\ t_3) && = l_1\\ &\quad \textbf{where}\ l_1 && = \mathit{At}\ t_1\ (\mathit{weave}\ l_1\ t_1)\ l_0\ l_1\ l_2\\ &\qquad\qquad l_2 && = \mathit{At}\ t_2\ (\mathit{weave}\ l_2\ t_2)\ l_0\ l_1\ l_3\\ &\qquad\qquad l_3 && = \mathit{At}\ t_3\ (\mathit{weave}\ l_3\ t_3)\ l_0\ l_2\ l_3\end{aligned}$$

Consider the definition of l_2 in the last case: it is labelled with t_2, going down recursively invokes *weave*, the *up* link is set to l_0, its left neighbour is l_1 and its right neighbour is l_3. This scheme generalizes in a straightforward manner to constructors of arbitrary arity. Note, however, that the definition of the locations is mostly independent of the particular constructor at hand. So, before we proceed, let us factor *weave* into a part that is specific to a particular term type and a part that is independent of it.

$$\begin{aligned}&\mathit{weave}\ l_0\ (\mathit{Var}\ s) && = \mathit{loc}_0\ \mathit{weave}\ l_0\\ &\mathit{weave}\ l_0\ (\mathit{Abs}\ s\ t_1) && = \mathit{loc}_1\ \mathit{weave}\ l_0\ t_1\\ &\mathit{weave}\ l_0\ (\mathit{App}\ t_1\ t_2) && = \mathit{loc}_2\ \mathit{weave}\ l_0\ t_1\ t_2\\ &\mathit{weave}\ l_0\ (\mathit{If}\ t_1\ t_2\ t_3) && = \mathit{loc}_3\ \mathit{weave}\ l_0\ t_1\ t_2\ t_3\\ &\mathit{loc}_0\ \mathit{wv}\ l_0 && = l_0\\ &\mathit{loc}_1\ \mathit{wv}\ l_0\ t_1 && = l_1\\ &\quad \textbf{where}\ l_1 && = \mathit{At}\ t_1\ (\mathit{wv}\ l_1\ t_1)\ l_0\ l_1\ l_1\end{aligned}$$

$$
\begin{array}{lll}
loc_2\ wv\ l_0\ t_1\ t_2 & = & l_1 \\
\quad \mathbf{where}\ l_1 & = & At\ t_1\ (wv\ l_1\ t_1)\ l_0\ l_1\ l_2 \\
\qquad\qquad l_2 & = & At\ t_2\ (wv\ l_2\ t_2)\ l_0\ l_1\ l_2 \\
loc_3\ wv\ l_0\ t_1\ t_2\ t_3 & = & l_1 \\
\quad \mathbf{where}\ l_1 & = & At\ t_1\ (wv\ l_1\ t_1)\ l_0\ l_1\ l_2 \\
\qquad\qquad l_2 & = & At\ t_2\ (wv\ l_2\ t_2)\ l_0\ l_1\ l_3 \\
\qquad\qquad l_3 & = & At\ t_3\ (wv\ l_3\ t_3)\ l_0\ l_2\ l_3
\end{array}
$$

Note that loc_m must be parameterized by the *weave* function so that it can be reused for different term types.

5 A Read-write Web

The Web introduced in the previous section is read-only since the links are created statically when *top* is called. So even if we change the subterm attached to a location, the change will not be remembered if we move onwards. To make the Web reflect any user edits, we must create the links dynamically as we move. To this end we turn the components of the type *Loc* into functions that create locations:

$$
\begin{array}{llll}
\mathbf{data}\ Loc\ a\ =\ At\{ & it & :: & a, \\
 & fdown & :: & a \to Loc\ a, \\
 & fup & :: & a \to Loc\ a, \\
 & fleft & :: & a \to Loc\ a, \\
 & fright & :: & a \to Loc\ a\}.
\end{array}
$$

The navigation primitives are implemented by calling the appropriate link function with the current subterm.

$$
\begin{array}{lll}
down, up, left, right & :: & Loc\ a \to Loc\ a \\
down\ l & = & (fdown\ l)\ (it\ l) \\
up\ l & = & (fup\ l)\ (it\ l) \\
left\ l & = & (fleft\ l)\ (it\ l) \\
right\ l & = & (fright\ l)\ (it\ l)
\end{array}
$$

The implementation of *weave* and loc_m is similar to what we had before except that any local changes are now propagated when we move (*weave* still does not have the right type).

$$
\begin{array}{lll}
top & = & fr\ \mathbf{where}\ fr\ t = At\ t\ (weave\ fr)\ fr\ fr\ fr \\
weave\ fl_0\ (Var\ s) & = & loc_0\ weave\ (fl_0\ (Var\ s)) \\
weave\ fl_0\ (Abs\ s\ t_1) & = & loc_1\ weave\ (\lambda t_1' \to fl_0\ (Abs\ s\ t_1'))\ t_1 \\
weave\ fl_0\ (App\ t_1\ t_2) & = & loc_2\ weave\ (\lambda t_1'\ t_2' \to fl_0\ (App\ t_1'\ t_2'))\ t_1\ t_2 \\
weave\ fl_0\ (If\ t_1\ t_2\ t_3) & = & loc_3\ weave\ (\lambda t_1'\ t_2'\ t_3' \to fl_0\ (If\ t_1'\ t_2'\ t_3'))\ t_1\ t_2\ t_3 \\
loc_0\ wv\ fl_0' & = & fl_0' \\
loc_1\ wv\ fl_0' & = & fl_1 \\
\quad \mathbf{where}\ fl_1\ t_1 & = & At\ t_1\ (wv\ (upd\ fl_1))\ (upd\ fl_0')\ (upd\ fl_1)\ (upd\ fl_1) \\
\qquad\qquad \mathbf{where}\ upd\ fl\ t_1' & = & fl\ t_1'
\end{array}
$$

$$
\begin{aligned}
&loc_2\ wv\ fl'_0 &&= fl_1\\
&\quad \textbf{where}\ fl_1\ t_1\ t_2 &&= At\ t_1\ (wv\ (upd\ fl_1))\ (upd\ fl'_0)\ (upd\ fl_1)\ (upd\ fl_2)\\
&\qquad\qquad \textbf{where}\ upd\ fl\ t'_1 &&= fl\ t'_1\ t_2\\
&\qquad\quad fl_2\ t_1\ t_2 &&= At\ t_2\ (wv\ (upd\ fl_2))\ (upd\ fl'_0)\ (upd\ fl_1)\ (upd\ fl_2)\\
&\qquad\qquad \textbf{where}\ upd\ fl\ t'_2 &&= fl\ t_1\ t'_2\\
&loc_3\ wv\ fl'_0 &&= fl_1\\
&\quad \textbf{where}\ fl_1\ t_1\ t_2\ t_3 &&= At\ t_1\ (wv\ (upd\ fl_1))\ (upd\ fl'_0)\ (upd\ fl_1)\ (upd\ fl_2)\\
&\qquad\qquad \textbf{where}\ upd\ fl\ t'_1 &&= fl\ t'_1\ t_2\ t_3\\
&\qquad\quad fl_2\ t_1\ t_2\ t_3 &&= At\ t_2\ (wv\ (upd\ fl_2))\ (upd\ fl'_0)\ (upd\ fl_1)\ (upd\ fl_3)\\
&\qquad\qquad \textbf{where}\ upd\ fl\ t'_2 &&= fl\ t_1\ t'_2\ t_3\\
&\qquad\quad fl_3\ t_1\ t_2\ t_3 &&= At\ t_3\ (wv\ (upd\ fl_3))\ (upd\ fl'_0)\ (upd\ fl_2)\ (upd\ fl_3)\\
&\qquad\qquad \textbf{where}\ upd\ fl\ t'_3 &&= fl\ t_1\ t_2\ t'_3
\end{aligned}
$$

To illustrate the propagation of changes consider the definition of fl_2 local to loc_3: it takes as arguments the three 'current' components t_1, t_2 and t_3 and creates a location labelled with the second component t_2. Now, if its *fright* function is called with, say, t'_2, then fl_3 is invoked with t_1, t'_2 and t_3 creating a new location labelled with t_3. If now *fup* t'_3 is called, fl'_0 is invoked with t_1, t'_2 and t'_3 as arguments. It in turn creates a new term and passes it to fl_0, the link function of its parent (see the definition of *weave*).

Finally, it is worth noting that all the primitives use constant time since they all reduce to a few function applications.

6 The Web Interface

The above implementation works very smoothly but it does not quite implement the interface given in Sec. 3. The last version of the weaver is defined by equations of the form

$$weave\ fl_0\ (K\ a_1\ \ldots\ a_n)\ =\ loc_m\ weave\ (\lambda t_1\ \ldots\ t_m \to fl_0\ (K\ a_1\ \ldots\ a_n))\ t_1\ \ldots\ t_m$$

whereas we want to let the user supply somewhat simpler equations of the form

$$weave\ (K\ a_1\ \ldots\ a_n)\ =\ con_m\ weave\ (\lambda t_1\ \ldots\ t_m \to K\ a_1\ \ldots\ a_n)\ t_1\ \ldots\ t_m.$$

Now, the second form can be obtained from the first if we flip the arguments of *weave* and split $\lambda t_1\ \ldots\ t_m \to fl_0\ (K\ a_1\ \ldots\ a_n)$ into the constructor function $\lambda t_1\ \ldots\ t_m \to K\ a_1\ \ldots\ a_n$ and the link function fl_0:

$$weave\ (K\ a_1\ \ldots\ a_n)\ fl_0\ =\ con_m\ weave\ (\lambda t_1\ \ldots\ t_m \to K\ a_1\ \ldots\ a_n)\ t_1\ \ldots\ t_m\ fl_0.$$

Applying η-reduction we obtain the desired form. Now, the combinators con_m must merely undo the flipping and splitting before they call loc_m.

$$
\begin{aligned}
&\textbf{newtype}\ Weaver\ a &&= W\{unW :: (a \to Loc\ a) \to Loc\ a\}\\
&call\ wv\ fl_0\ t &&= unW\ (wv\ t)\ fl_0
\end{aligned}
$$

$$\begin{array}{lcl}
con_0\ wv\ k & = & W\ (\lambda fl_0 \to loc_0\ (call\ wv)\ (fl_0\ k)) \\
con_1\ wv\ k\ t_1 & = & W\ (\lambda fl_0 \to loc_1\ (call\ wv)\ (\lambda t_1 \to fl_0\ (k\ t_1))\ t_1) \\
con_2\ wv\ k\ t_1\ t_2 & = & W\ (\lambda fl_0 \to loc_2\ (call\ wv)\ (\lambda t_1\ t_2 \to fl_0\ (k\ t_1\ t_2))\ t_1\ t_2) \\
con_3\ wv\ k\ t_1\ t_2\ t_3 & = & W\ (\lambda fl_0 \to loc_3\ (call\ wv)\ (\lambda t_1\ t_2\ t_3 \to fl_0\ (k\ t_1\ t_2\ t_3))\ t_1\ t_2\ t_3)
\end{array}$$

Note that we have also taken the opportunity to introduce a new type for weavers that hides the implementation from the user. It remains to define *explore*:

$$explore\ wv\ =\ fr\ \mathbf{where}\ fr\ t = At\ t\ (call\ wv\ fr)\ fr\ fr\ fr.$$

Finally, note that the Web no longer relies on lazy evaluation since the only recursively defined objects are functions.

Exercise 1
Write a function $con :: (a \to Weaver\ a) \to ([a] \to a) \to ([a] \to Weaver\ a)$ that generalizes the con_m combinators. Instead of taking m components as separate arguments it takes a list of components.

References

Huet, G. (1997) Functional Pearl: The Zipper. *J. Functional Programming* **7**(5):549–554.

Jones, M. and Peterson, J. (1999) *Hugs 98 User Manual.* Available from `http://www.haskell.org/hugs`.

Peyton Jones, S. and Hughes, J. (eds). (1999) *Haskell 98 — A Non-strict, Purely Functional Language.* Available from `http://www.haskell.org/definition/`.